Higher-Order Linearisability*

Andrzej S. Murawski[1] and Nikos Tzevelekos[2]

1 **University of Warwick, Coventry, UK**
2 **Queen Mary University of London, London, UK**

Abstract

Linearisability is a central notion for verifying concurrent libraries: a library is proven correct if its operational history can be rearranged into a sequential one that satisfies a given specification. Until now, linearisability has been examined for libraries in which method arguments and method results were of ground type. In this paper we extend linearisability to the general higher-order setting, where methods of arbitrary type can be passed as arguments and returned as values, and establish its soundness.

1998 ACM Subject Classification F.1.2 Models of Computation

Keywords and phrases Linearisability, Concurrency, Higher-Order Computation

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.34

1 Introduction

Software libraries provide implementations of routines, often of specialised nature, to facilitate code reuse and modularity. To support the latter, they should follow specifications that describe the range of acceptable behaviours for correct and safe deployment. Adherence to specifications can be formalised using the classic notion of contextual approximation (refinement), which scrutinises the behaviour of code in any possible context. Unfortunately, the quantification makes it difficult to prove contextual approximations directly, which motivates research into sound techniques for establishing it.

In the concurrent setting, a notion that has been particularly influential is that of *linearisability* [12]. Linearisability requires that, for each history generated by a library, one should be able to find another history from the specification (a *linearisation*), which matches the former up to certain rearrangements of events. In the original formulation by Herlihy and Wing [12], these permutations were not allowed to disturb the order between library returns and client calls. Moreover, linearisations were required to be *sequential* traces, that is, sequences of method calls immediately followed by their returns.

In this paper we shall work with *open higher-order* libraries, which provide implementations of *public* methods and may themselves depend on *abstract* ones, to be supplied by parameter libraries. The classic notion of linearisability only applies to closed libraries (without abstract methods). Additionally, both method arguments and results had to be of *ground* type. The closedness limitation was recently lifted in [13, 3], which distinguished between public (or *implemented*) and abstract methods (*callable*). Although [13] did not in principle exclude higher-order functions, those works focussed on linearisability for the case where the allowable methods were restricted to first-order functions ($\mathsf{int} \to \mathsf{int}$). Herein, we give a systematic exposition of linearisability for general higher-order concurrent libraries,

* Research was partially supported by the EPSRC (EP/P004172/1).

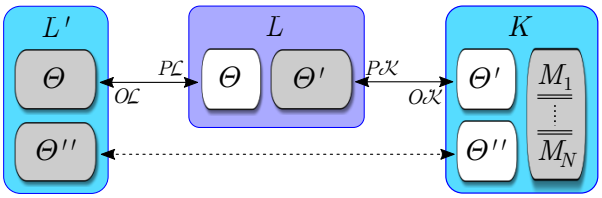

Figure 1 A library $L : \Theta \to \Theta'$ in environment comprising a parameter library $L' : \varnothing \to \Theta, \Theta''$ and a client K of the form $\Theta', \Theta'' \vdash M_1 \| \cdots \| M_N$.

where methods can be of arbitrary higher-order types. In doing so, we also propose a corresponding notion of sequential history for higher-order library interactions.

We examine libraries L that can interact with their environments by means of public and abstract methods: a library L with abstract methods of types $\Theta = \theta_1, \cdots, \theta_n$ and public methods $\Theta' = \theta'_1, \cdots, \theta'_{n'}$ is written as $L : \Theta \to \Theta'$. We shall work with arbitrary higher-order types generated from the ground types unit and int. Types in Θ, Θ' must always be function types, i.e. their order is at least 1.

A library L may be used in computations by placing it in a context that will keep on calling its public methods (via a client K) as well as providing implementations for the abstract ones (via a parameter library L'). The setting is depicted in Figure 1. Note that, as the library L interacts with K and L', they exchange functions between each other. Consequently, in addition to K making calls to public methods of L and L making calls to its abstract methods, K and L' may also issue calls to functions that were passed to them as arguments during higher-order interactions. Analogously, L may call functions that were communicated to it via library calls.

Our framework is operational in flavour and draws upon concurrent [15, 7] and operational game semantics [14, 16, 8]. We shall model library use as a game between two participants: *Player (P)*, corresponding to the library L, and *Opponent (O)*, representing the environment (L', K) in which the library was deployed. Each call will be of the form $\mathsf{call}\, m(v)$ with the corresponding return of the shape $\mathsf{ret}\, m(v)$, where v is a value. As we work in a higher-order framework, v may contain functions, which can participate in subsequent calls and returns. Histories will be sequences of *moves*, which are calls and returns paired with thread identifiers. A history is sequential just if every move produced by O is immediately followed by a move by P in the same thread. In other words, the library immediately responds to each call or return delivered by the environment. In contrast to classic linearisability, the move by O and its response by P need not be a call/return pair, as the higher-order setting provides more possibilities (in particular, the P response may well be a call). Accordingly, linearisable higher-order histories can be seen as sequences of atomic segments (linearisation points), starting at environment moves and ending with corresponding library moves.

In the spirit of [3], we are going to consider two scenarios: one in which K and L' share an explicit communication channel (the general case) as well as a situation in which they can only communicate through the library (the encapsulated case). Further, we also handle the case in which extra closure assumptions can be made about the parameter library (the relational case), which can be useful for dealing with a variety of assumptions on the use of parameter libraries that may arise in practice. In each case, we present a candidate definition of linearisability and illustrate it with tailored examples. The suitability of each kind of linearisability is demonstrated by showing that it implies the relevant form of contextual approximation (refinement). We also examine compositionality of the proposed concepts.

One of our examples will discuss the implementation of the flat-combining approach [11, 3], adapted to higher-order types.

1.1 Example: a higher-order multiset library

Higher-order libraries are common in languages like ML, Java, Python, etc. As an illustrative example, we consider a library written in ML-like syntax which implements a multiset data structure with integer elements. For simplicity, we assume that its signature contains just two methods:

$$\mathit{count} : \mathsf{int} \to \mathsf{int}, \quad \mathit{update} : (\mathsf{int} \times (\mathsf{int} \to \mathsf{int})) \to \mathsf{int}\,.$$

The former method returns for each integer its multiplicity in the multiset – this is 0 if the integer is not a member of the multiset. On the other hand, *update* takes as an argument an integer i and a function f, and updates the multiplicity of i in the multiset to $|f(i)|$ (we use the absolute value of $f(i)$ in order to meet the multiset requirement that element multiplicities not be negative; alternatively, we could have used exceptions to quarantine such client method behaviour). Methods with the same functionalities can be found e.g. in the multiset module of the ocaml-containers library [1]. While our example is simple, the same kind of analysis as below can be applied to more intricate examples such as *map* methods for integer-valued arrays, maps or multisets.

▶ **Example 1** (Multiset)**.** Consider the concurrent multiset library L_{mset} in Figure 2. It uses a private reference for storing the multiset's characteristic function and reads *optimistically*, without locking (cf. [10, 19]). The *update* method in particular reads the current multiplicity of the given element i (via *count*) and computes its new multiplicity without acquiring a lock on the characteristic function. It only acquires a lock when it is ready to write the new value (line 10) in the hope that the value at i will still be the same and the update can proceed; if not, another attempt to update the value is made.

Let us look at some example executions of the library via their resulting histories, i.e. sequences of method calls and returns between the library and a client. In the topmost block (a) of history diagrams of Figure 2, we see three such executions. Note that we do not record internal calls to *count* or *aux*, and use m and variants for method identifiers (names). We use the abbreviation *cnt* for *count*, and *upd* for *update*, and initially ignore the circled events for *cnt*. Each execution involves 2 threads.

In the first execution, the client calls $\mathit{update}(i, m)$ in the second thread, and subsequently calls $\mathit{count}(i)$ in the first thread. The code for *update* stipulates that first $\mathit{count}(i)$ be called internally, returning some multiplicity j for i, and then $m(j)$ should be called. As soon m returns a value j', *update* sets the multiplicity of i to j' and itself returns j'. The last event in this history is a return of *count* in the first thread with the old value j. According to our proposed definition, this history will be linearisable to another, intuitively correct one: the last return can be moved to the circled position. At this point the notion of linearisability is used informally, but it will be made precise in the following sections. In the second execution, the last return of *count* in the first thread returns the updated value. In this case, we will be able to move call $\mathit{cnt}(i)$ to the circled position to obtain a linearisation, which is obviously correct. Finally, in the third execution we have a history that will turn out non-linearisable to an intuitively correct history. Indeed, we should not be able to return the updated value in the first thread before m has returned it in the second one.

The two histories in block (b) in the same figure demonstrate the mechanism for updates. The first history will be linearisable to the second one. In the second history we see that both

```
1  public count, update;
2  Lock lock;
3  F := λx.0;
4
5  count = λi. (!F)i
6  update = λ(i, g). aux(i, g, count i)
7
8  aux = λ(i, g, j).
9    let y = |g j| in
10     lock.acquire();
11     let f = !F in
12       if (j == (f i)) then {
13         F := λx. if (x == i) then y
14                  else (f x) ;
15         lock.release();
16         y }
17       else {
18         lock.release();
19         aux(i, g, f i) }
```

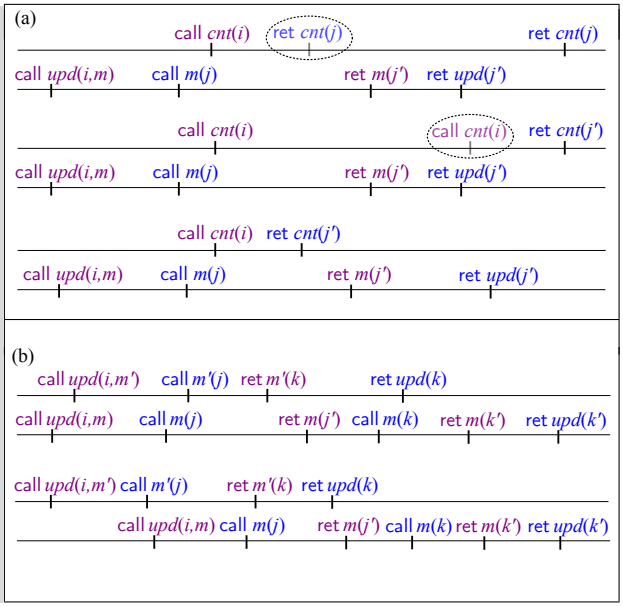

Figure 2 Multiset library L_{mset} with public methods $count : \mathsf{int} \to \mathsf{int}$ and $update : \mathsf{int} \times (\mathsf{int} \to \mathsf{int}) \to \mathsf{int}$.

threads try to update the same element i, but the first one succeeds in it first and returns k on *update*. Then, the second thread realises that the value of i has been updated to k and calls m again, this time with argument k. An important feature of the second history is that it is *sequential*: each client event (call or return) is immediately followed by a library event.

Observe that the rearrangements discussed above involve either advancing a library action or postponing an environment action and that each action could be a call or a return. Definition 6 will capture this formally. For now, we note that this generalises the classic setting [12], where library method returns could be advanced and environment method calls deferred.

2 Higher-order linearisability

We examine higher-order libraries interacting with their context by means of abstract and public methods. In particular, we shall rely on types given by the grammar on the left below. We let Meths stand for the set of *method names* and assume $\mathsf{Meths} = \biguplus_{\theta,\theta'} \mathsf{Meths}_{\theta,\theta'}$, where each set $\mathsf{Meths}_{\theta,\theta'}$ contains names for methods of type $\theta \to \theta'$. Methods are ranged over by m (and variants). We let v range over computational *values*, which include a unit value, integers, methods and pairs of values.

$$\theta ::= \mathsf{unit} \mid \mathsf{int} \mid \theta \times \theta \mid \theta \to \theta \qquad v ::= () \mid i \mid m \mid (v, v)$$

The framework of a higher-order library and its environment is depicted in Figure 1. Given $\Theta, \Theta' \subseteq \mathsf{Meths}$, a library L is said to have type $\Theta \to \Theta'$ if it defines public methods with names (and types) as in Θ', using abstract methods Θ. The environment of L consists of a *client* K (which invokes public methods of Θ'), and a *parameter library* L' (which provides code for the abstract methods Θ). In general, K and L' may interact via a disjoint set of methods $\Theta'' \subseteq \mathsf{Meths}$, to which L has no access.

In the rest of this paper, we shall implicitly assume that we work with a library L operating in an environment presented in Figure 1. The client K will consist of a fixed number N of concurrent threads. Next we introduce a notion of history tailored to the setting and define how histories can be linearised. In Section 3 we present the syntax for libraries and clients, and in Section 4 we define their semantics in terms of histories and co-histories respectively.

2.1 Higher-order histories

The operational semantics of libraries will be given in terms of *histories*, which are sequences of method calls and returns, each decorated with a thread identifier t and a *polarity index* XY, where $X \in \{O, P\}$ and $Y \in \{\mathcal{L}, \mathcal{K}\}$, as shown below.

$$(t, \mathsf{call}\, m(v))_{XY} \qquad (t, \mathsf{ret}\, m(v))_{XY}$$

We shall refer such decorated calls and returns as ***moves***. Here, m is a method name and v is a value of a matching type. The index XY specifies which of the three entities (L, L', K) produces the move, and towards whom it is addressed.

- If $XY = P\mathcal{L}$ then the move is issued by L, and is addressed to L'.
- If $XY = P\mathcal{K}$ then the move is issued by L, and is addressed to K.
- If $XY = O\mathcal{L}$ then the move is issued by L', and is addressed to L.
- If $XY = O\mathcal{K}$ then the move is issued by K, and is addressed to L.

The choice of indices is motivated by the fact that the moves can be seen as defining a 2-player game between the library (L), which represents the *Proponent* player in the game (P), and its environment (L', K) that represents the *Opponent* (O). Moves played between L and L' are moreover decorated with $\mathcal{L}$, whereas those between L and K have $\mathcal{K}$ instead. Note that any potential interaction between L' and K is invisible to L and is therefore not accounted for in the game (but we will later see how it can affect it). We use O to refer to either $O\mathcal{K}$ or $O\mathcal{L}$, and P to refer to either $P\mathcal{K}$ or $P\mathcal{L}$. Finally, we let the ***dual*** polarity of XY to be $X'Y$, where $X \neq X'$. For example, the dual of $P\mathcal{L}$ is $O\mathcal{L}$.

Next we proceed to define histories. Their definition will rely on a more primitive concept of *prehistories*, which are sequences of method calls and returns that respect a stack discipline.

▶ **Definition 2.** *Prehistories* are sequences generated by one of the grammars:

$$\begin{array}{lcl} \mathsf{PreH}_O & ::= & \epsilon \mid \mathsf{call}\, m(v)_{OY}\ \mathsf{PreH}_P\ \mathsf{ret}\, m(v')_{PY}\ \mathsf{PreH}_O \\ \mathsf{PreH}_P & ::= & \epsilon \mid \mathsf{call}\, m(v)_{PY}\ \mathsf{PreH}_O\ \mathsf{ret}\, m(v')_{OY}\ \mathsf{PreH}_P \end{array}$$

where, in each line, the two occurrences of $Y \in \{\mathcal{K}, \mathcal{L}\}$ and $m \in \mathsf{Meths}$ must each match. Moreover, if $m \in \mathsf{Meths}_{\theta,\theta'}$, the types of v, v' must match θ, θ' respectively. We let $\mathsf{PreH} = \mathsf{PreH}_O \cup \mathsf{PreH}_P$.

Thus, prehistories from PreH_O start with an O-move, while those in PreH_P start with a P-move. In each case, the polarities inside a prehistory alternate between O and P, and the polarities of calls and matching returns are always dual (*returns dual to calls*). For example, a call made by L to L' (tagged $P\mathcal{L}$) must be matched by a return from L' to L (tagged $O\mathcal{L}$).

Histories will be interleavings of prehistories tagged with thread identifiers (natural numbers), subject to a set of well-formedness constrains. In particular, a history h for library $L : \Theta \to \Theta'$ will have to begin with an O-move and satisfy the following conditions, to be formalised in Definition 3.

1. The name of any method called in h must come from Θ or Θ', or be introduced earlier in h as a higher-order argument or result (*no methods out of thin air*). In addition:
   - if the method is from Θ', the call must be tagged with $O\mathcal{K}$ (i.e. issued by K);
   - if the method is from Θ, the call must be tagged with $P\mathcal{L}$ (i.e. issued by L towards L');
   - for a call of method $m \notin \Theta \cup \Theta'$ to be valid, m must be introduced in an earlier move of dual polarity (*calls dual to introductions*).
2. Any method name appearing inside a call or return argument in h must be *fresh*, i.e. not used earlier. This reflects the assumption that methods can be called and returned, but not compared for identity (*introductions always fresh*).

Given $h \in \mathsf{PreH}$ and $t \in \mathbb{N}$, we write $t \times h$ for h in which each call or return is decorated with t. We refer to such moves with $(t, \mathsf{call}\, m(v))_{XY}$ or $(t, \mathsf{ret}\, m(v))_{XY}$ respectively. If we only want to stress the X or Y membership, we shall drop Y or X respectively. Moreover, when no confusion arises, we may sometimes drop a move's polarity altogether. We say that a move x ***introduces*** a name $m \in \mathsf{Meths}$ when $x \in \{\mathsf{call}\, m'(v), \mathsf{ret}\, m'(v)\}$ for some m', v such that v contains m.

▶ **Definition 3.** Given Θ, Θ', the set of ***histories*** over $\Theta \to \Theta'$, written $\mathcal{H}_{\Theta,\Theta'}$, is defined by

$$\mathcal{H}_{\Theta,\Theta'} = \bigcup_{N>0} \bigcup_{h_1,\cdots,h_N \in \mathsf{PreH}_O} (1 \times h_1) \mid \cdots \mid (N \times h_N)$$

where $(1 \times h_1) \mid \cdots \mid (N \times h_N)$ is the set of all interleavings of $(1 \times h_1), \cdots, (N \times h_N)$ satisfying:

1. For any $s_1(t, \mathsf{call}\, m(v))_{XY} s_2 \in \mathcal{H}_{\Theta,\Theta'}$, either $m \in \Theta'$ and $XY = O\mathcal{K}$, or $m \in \Theta$ and $XY = P\mathcal{L}$, or there is a move $(t', x)_{X'Y}$ in s_1 such that $X \neq X'$ and x introduces m.
2. For any $s_1(t, x)_{XY} s_2 \in \mathcal{H}_{\Theta,\Theta'}$ and any m, if m is introduced by x then m must not occur in s_1.

A history $h \in \mathcal{H}_{\Theta,\Theta'}$ is called ***sequential*** if it is of the form

$$h = (t_1, x_1)_{OY_1} (t_1, x_1')_{PY_1'} \cdots (t_k, x_k)_{OY_k} (t_k, x_k')_{PY_k'}$$

for some $t_i, x_i, x_i', Y_i, Y_i'$. We write $\mathcal{H}^{\mathsf{seq}}_{\Theta,\Theta'}$ for the set of all sequential histories from $\mathcal{H}_{\Theta,\Theta'}$.

We shall range over $\mathcal{H}_{\Theta,\Theta'}$ using h, s (and variants). The subscripts Θ, Θ' will often be omitted. Given a history h, we shall write $\overline{h}$ for the sequence of moves obtained from h by dualising all move polarities inside it. The set of ***co-histories*** over $\Theta \to \Theta'$ will be $\mathcal{H}^{co}_{\Theta,\Theta'} = \{\overline{h} \mid h \in \mathcal{H}_{\Theta,\Theta'}\}$.

While in this section histories will be extracted from example libraries informally, in Section 4 we give the formal semantics $[\![L]\!]$ of libraries. For each $L : \Theta \to \Theta'$, we shall have $[\![L]\!] \subseteq \mathcal{H}_{\Theta,\Theta'}$.

▶ Remark 4. The notion of history introduced above extends the classic notion from [12] to higher-order types. It also extends the notion presented in [3]. The intuition behind the definition is that a history is a sequence of (well-bracketed) method calls and returns, called *moves*, each tagged with a thread identifier and a polarity, where polarities track the originators and recipients of moves. Moves may be calls or returns related to methods given in the library interface ($\Theta \to \Theta'$), or dynamically created methods that appear earlier inside the histories – recall that, in a higher-order setting, methods can be passed around as arguments to calls or be returned as results by other methods. On the other hand, a sequential history is one in which the operations performed by the library can be perceived as **atomic**, that is, each move produced by O is to be immediately followed by the library's response, which is a P move in the same thread.

▶ **Example 5** (Multiset spec)**.** We now revisit our first example and provide a specification for it. Recall the multiset library L_{mset} from Figure 2. Our verification goal will be to prove linearisability of L_{mset} to a specification $A_{\mathsf{mset}} \subseteq \mathcal{H}^{\mathsf{seq}}_{\varnothing,\Theta}$, where $\Theta = \{\mathit{count}, \mathit{update}\}$, which we define below. A_{mset} will certify that L_{mset} correctly implements some integer multiset I whose elements change over time according to the moves in h. For a multiset I and natural numbers i, j, we write $I(i)$ for the multiplicity of i in I, and $I[i \mapsto j]$ for I with its multiplicity of i set to j. We shall stipulate that moves inside histories $h \in A_{\mathsf{mset}}$ be annotatable with multisets I in such a way that the multiset is empty at the start of h (i.e. $I(i) = 0$ for all i) and:

- If I is changed between two consecutive moves in h then the second move is a P-move. In other words, the client cannot directly update the elements of I.
- Each call to *count* on argument i must be immediately followed by a return with value $I(i)$, and with I remaining unchanged.
- Each call to *update* on (i, m) must be followed by a call to m on $I(i)$, with I unchanged. Moreover, m must later return with some value j. Assuming at that point the multiset will have value J, if $I(i) = J(i)$ then the next move is a return of the original *update* call, with value j; otherwise, a new call to m on $J(i)$ is produced, and so on.

We formally define the specification next.

Let $\mathcal{H}^{\circ}_{\varnothing,\Theta}$ contain sequences of moves from $\varnothing \to \Theta$ accompanied by a multiset (i.e. the sequences consist of elements of the form $(t, x, I)_{XY}$). For each $s \in \mathcal{H}^{\circ}_{\varnothing,\Theta}$, we let $\pi_1(s)$ be the history extracted by projection, i.e. $\pi_1(s) \in \mathcal{H}_{\varnothing,\Theta}$. For each t, we let $s \upharpoonright t$ be the subsequence of s of elements with first component t. Writing $\sqsubseteq_{\mathrm{pre}}$ for the prefix relation, and dropping the Y index from moves (Y is always $\mathcal{K}$ here), we define $A_{\mathsf{mset}} = \{\pi_1(s) \mid s \in A^{\circ}_{\mathsf{mset}}\}$ where:

$$A^{\circ}_{\mathsf{mset}} = \{\ s \in \mathcal{H}^{\circ}_{\varnothing,\Theta} \ \mid\ \pi_1(s) \in \mathcal{H}^{\mathsf{seq}}_{\varnothing,\Theta} \ \wedge\ \forall t.\, s \upharpoonright t \in \mathcal{S} \ \wedge\ \forall s'(\_, I)_P(\_, J)_O \sqsubseteq_{\mathrm{pre}} s.\ I = J\ \}$$

and, for each t, the set of t-indexed annotated histories $\mathcal{S}$ is given by the following grammar:

$$\begin{aligned}
\mathcal{S} \ &\to \epsilon \ \mid\ (t, \mathsf{call}\ \mathit{cnt}(i), I)_O\, (t, \mathsf{ret}\ \mathit{cnt}(I(i)), I)_P\, \mathcal{S} \\
&\quad \mid\ (t, \mathsf{call}\ \mathit{upd}(i, m), I)_O\, \mathcal{M}^{i,j}_{I,J}\, (t, \mathsf{ret}\ \mathit{upd}(|j|), J[i \mapsto |j|])_P\, \mathcal{S} \\
\mathcal{M}^{i,j}_{I,J} &\to (t, \mathsf{call}\, m(I(i)), I)_P\, \mathcal{S}\, (t, \mathsf{ret}\, m(j), J)_O \qquad \text{provided } J(i) = I(i) \\
\mathcal{M}^{i,j}_{I,J} &\to (t, \mathsf{call}\, m(I(i)), I)_P\, \mathcal{S}\, (t, \mathsf{ret}\, m(j'), J')_O\, \mathcal{M}^{i,j}_{J',J} \qquad \text{provided } J'(i) \neq I(i)
\end{aligned}$$

By definition, all histories in A_{mset} are sequential. The elements of $A^{\circ}_{\mathsf{mset}}$ carry along the multiset I that is being represented. The conditions on $A^{\circ}_{\mathsf{mset}}$ stipulate that O cannot change the value of I, while the rest of the conditions above are imposed by the grammar for $\mathcal{S}$. With the notion of linearisability to be introduced next, we will be able to show that $[\![L_{\mathsf{mset}}]\!]$ is indeed linearisable to A_{mset}.

2.2 Three notions of linearisability

We present three notions of linearisability. First introduce a general notion that generalises classic linearisability [12] and parameterised linearisability [3]. We then develop two more specialised variants: a notion of encapsulated linearisability, following [3], that captures scenarios where the parameter library and the client cannot directly interact; and a relational notion whereby context behaviour (client and parameter library) is known to be relationally invariant.

We begin by introducing a class of reorderings on histories. Suppose $X, X' \in \{O, P\}$ and $X \neq X'$. We let $\lhd_{XX'} \subseteq \mathcal{H}_{\Theta,\Theta'} \times \mathcal{H}_{\Theta,\Theta'}$ be the smallest binary relation over $\mathcal{H}_{\Theta,\Theta'}$ satisfying, for any $t \neq t'$:

$$s_1(t', x')_{Z'}(t, x)_Z\, s_2 \ \lhd_{XX'}\ s_1(t, x)_Z(t', x')_{Z'} s_2$$

whenever $Z = X$ or $Z' = X'$. Intuitively, two histories h_1, h_2 are related by $\lhd_{XX'}$ if the latter can be obtained from the former by swapping two adjacent moves from different threads in such a way that, after the swap, an X-move will occur earlier or an X'-move will occur later. Note that, because of $X \neq X'$, the relation always applies to adjacent moves of the same polarity. On the other hand, we cannot have $s_1(t,x)_X(t',x')_{X'}s_2 \lhd_{XX'} s_1(t',x')_{X'}(t,x)_X s_2$.

▶ **Definition 6** (General Linearisability)**.** Given $h_1, h_2 \in \mathcal{H}_{\Theta,\Theta'}$, we say that h_1 ***is linearised by*** h_2, written $h_1 \sqsubseteq h_2$, if $h_1 \lhd_{PO}^* h_2$. Given libraries $L, L' : \Theta \to \Theta'$ and a set of sequential histories $A \subseteq \mathcal{H}^{\mathsf{seq}}_{\Theta,\Theta'}$ we write $L \sqsubseteq A$, and say that L *can be linearised to* A, if for any $h \in [\![L]\!]$ there exists $h' \in A$ such that $h \sqsubseteq h'$. Moreover, we write $L \sqsubseteq L'$ if $L \sqsubseteq [\![L']\!] \cap \mathcal{H}^{\mathsf{seq}}_{\Theta,\Theta'}$ (i.e. for all $h \in [\![L]\!]$ there is sequential $h' \in [\![L']\!]$ such that $h \sqsubseteq h'$).

▶ Remark 7. The classic notion of linearisability from [12] states that h linearises to h' just if the return/call order of h is preserved in h' (and h' is sequential), i.e. if a return move precedes a call move in h then so is the case in h'. Observing that, in [12], return and call moves coincide with P- and O-moves respectively, we can see that our higher-order notion of linearisability is a generalisation of the classic notion.

We next show that a more permissive notion of linearisability applies if the parameter library L' of Figure 1 is encapsulated, that is, the client K can have no direct access to it (i.e. $\Theta'' = \varnothing$). To capture the more restrictive nature of interaction, we introduce a more constrained notion of a history. Specifically, in addition to sequentiality in every thread, we shall insist that a move made by the library in the $\mathcal{L}$ or $\mathcal{K}$ component must be followed by an O move from the *same* component.

▶ **Definition 8.** We call a history $h \in \mathcal{H}_{\Theta,\Theta'}$ ***encapsulated*** if, for each thread t, we have that if $h = s_1\,(t,x)_{PY}\,s_2\,(t,x')_{OY'}\,s_3$ and moves from t are absent from s_2 then $Y = Y'$. Moreover, we set $\mathcal{H}^{\mathsf{enc}}_{\Theta,\Theta'} = \{h \in \mathcal{H}_{\Theta,\Theta'} \mid h \text{ encapsulated}\}$ and $[\![L]\!]_{\mathrm{enc}} = [\![L]\!] \cap \mathcal{H}^{\mathsf{enc}}_{\Theta,\Theta'}$ (if $L : \Theta \to \Theta'$).

We define the corresponding linearisability notion as follows. First, let $\diamond \subseteq \mathcal{H}_{\Theta,\Theta'} \times \mathcal{H}_{\Theta,\Theta'}$ be the smallest binary relation on $\mathcal{H}_{\Theta,\Theta'}$ such that, for any $Y, Y' \in \{\mathcal{K}, \mathcal{L}\}$ with $Y \neq Y'$ and $t \neq t'$:

$$s_1(t,m)_Y(t',m')_{Y'}s_2 \;\diamond\; s_1(t',m')_{Y'}(t,m)_Y s_2$$

▶ **Definition 9** (Encapsulated linearisability)**.** Given $h_1, h_2 \in \mathcal{H}^{\mathsf{enc}}_{\Theta,\Theta'}$, we say that h_1 is ***enc-linearised*** by h_2, and write $h_1 \sqsubseteq_{\mathrm{enc}} h_2$, if $h_1(\lhd_{PO} \cup \diamond)^* h_2$ and h_2 is sequential. A library $L : \Theta \to \Theta'$ can be *enc-linearised to* A, written $L \sqsubseteq_{\mathrm{enc}} A$, if $A \subseteq \mathcal{H}^{\mathsf{seq}}_{\Theta,\Theta'} \cap \mathcal{H}^{\mathsf{enc}}_{\Theta,\Theta'}$ and for any $h \in [\![L]\!]_{\mathrm{enc}}$ there exists $h' \in A$ such that $h \sqsubseteq_{\mathrm{enc}} h'$. We write $L \sqsubseteq_{\mathrm{enc}} L'$ if $L \sqsubseteq_{\mathrm{enc}} [\![L']\!]_{\mathrm{enc}} \cap \mathcal{H}^{\mathsf{seq}}_{\Theta,\Theta'}$.

▶ Remark 10. Suppose $\Theta = \{m : \mathsf{int} \to \mathsf{int}\}$ and $\Theta' = \{m' : \mathsf{int} \to \mathsf{int}\}$. Histories from $\mathcal{H}_{\Theta,\Theta'}$ may contain the following actions only: $\mathsf{call}\, m'(i)_{O\mathcal{K}}$, $\mathsf{ret}\, m(i)_{O\mathcal{L}}$, $\mathsf{call}\, m(i)_{P\mathcal{L}}$, $\mathsf{ret}\, m'(i)_{P\mathcal{K}}$. Then $(\lhd_{PO} \cup \diamond)^*$ preserves the order between $\mathsf{call}\, m(i)_{P\mathcal{L}}$ and $\mathsf{ret}\, m(i)_{O\mathcal{L}}$ as well as that between $\mathsf{ret}\, m'(i)_{P\mathcal{K}}$ and $\mathsf{call}\, m'(i)_{O\mathcal{K}}$, i.e. it coincides with Definition 3 of [3].

▶ **Example 11** (Parameterised multiset)**.** We revisit the multiset library of Example 1 and extend it with a public method *reset*, which performs multiplicity resets to default values using an abstract method *default* as the default-value function (again, we use absolute values to avoid negative multiplicities). The extended library is shown in Figure 3 and written $L_{\mathsf{mset2}} : \{\mathit{default}\} \to \Theta'$, with $\Theta' = \{\mathit{count},\ \mathit{update},\ \mathit{reset}\}$. In contrast to the *update* method of L_{mset}, *reset* is not optimistic: it retrieves the lock upon its call, and only releases it before return. In particular, the method calls *default* while it retains the lock.

```
1  public count, update, reset;
2  abstract default;
3  Lock lock;
4  F := λx.0;
5  ...
20 reset = λi.
21   lock.acquire();
22   let y = |default i| in
23     let f = !F in
24       F := λx. if (x == i) then y
25                else (f x);
26       lock.release();
27       y
```

```
1  public run; ...;
2  Lock lock;
3  struct {fun, arg, wait, retv} requests[N];
4
5  run = λ (f,x).
6    requests[t_id].fun := f;
7    requests[t_id].arg := x;
8    requests[t_id].wait := 1;
9    while (requests[t_id].wait)
10     if (lock.tryacquire()) {
11       for (t=0; t<N; t++)
12         if (requests[t].wait) {
13           requests[t].retv :=
14                 requests[t].fun (requests[t].arg);
15           requests[t].wait := 0;
16         }; lock.release() };
17   requests[t_id].retv;
```

Figure 3 Left: Parameterised multiset library L_{mset2} (lines 5-19 as in Fig. 2) with public methods *count*, *reset*: $\mathsf{int}\to\mathsf{int}$, *update*: $\mathsf{int}\times(\mathsf{int}\to\mathsf{int})\to\mathsf{int}$; abstract method *default*: $\mathsf{int}\to\mathsf{int}$. Right: Flat combination library L_{fc}.

Observe that, were *default* able to externally call *update*, we would reach a deadlock: *default* would be keeping the lock while waiting for the return of a method that requires the lock. On the other hand, if the library is encapsulated then the latter scenario is not possible. In such a case, L_{mset2} linearises to the specification A_{mset2}, defined next. Let $A_{\mathsf{mset2}} = \{\pi_1(s) \mid s \in A^{\circ}_{\mathsf{mset2}}\}$ where:

$$A^{\circ}_{\mathsf{mset2}} = \{\ s \in \mathcal{H}^{\circ}_{\emptyset,\Theta'} \ \mid\ \pi_1(s) \in \mathcal{H}^{\mathsf{seq}}_{\emptyset,\Theta'} \ \wedge\ \forall t.\, s \upharpoonright t \in \mathcal{S} \ \wedge\ \forall s'(\_, I)_P(\_, J)_O \sqsubseteq_{\mathrm{pre}} s.\, I = J\ \}$$

and the set $\mathcal{S}$ is now given by the grammar of Example 5 extended with the rule:

$$\mathcal{S} \to\ (t, \mathsf{call}\ \mathit{reset}(i), I)_{OK}\,(t, \mathsf{call}\ \mathit{default}(i), I)_{P\mathcal{L}}\,(t, \mathsf{ret}\ \mathit{default}(j), I)_{O\mathcal{L}}\,(t, \mathsf{ret}\ \mathit{reset}(|j|), I')_{PK}\,\mathcal{S}$$

with $I' = I[i \mapsto |j|]$. Our framework makes it possible to confirm that L_{mset2} enc-linearises to A_{mset2}.

We finally extend general linearisability to cater for situations where the client and the parameter library adhere to closure constraints expressed by relations $\mathcal{R}$ on histories. Let Θ, Θ' be sets of abstract and public methods respectively. The closure relations we consider are closed under permutations of methods outside $\Theta \cup \Theta'$: if $h\,\mathcal{R}\,h'$ and π is a (type-preserving) permutation on $\mathsf{Meths} \setminus (\Theta \cup \Theta')$ then $\pi(h)\,\mathcal{R}\,\pi(h')$. The requirement represents the fact that, apart from the method names from a library interface, the other method names are arbitrary and can be freely permuted without any observable effect. Thus, $\mathcal{R}$ should not be distinguishing between such names.

▶ **Definition 12** (Relational linearisability). Let $\mathcal{R} \subseteq \mathcal{H}_{\Theta,\Theta'} \times \mathcal{H}_{\Theta,\Theta'}$ be closed under permutations of names in $\mathsf{Meths} \setminus (\Theta \cup \Theta')$. Given $h_1, h_2 \in \mathcal{H}_{\Theta,\Theta'}$, we say that h_1 is ***$\mathcal{R}$-linearised*** by h_2, and write $h_1 \sqsubseteq_{\mathcal{R}} h_2$, if $h_1(\lhd_{PO} \cup \mathcal{R})^* h_2$ and h_2 is sequential. A library $L : \Theta \to \Theta'$ can be *$\mathcal{R}$-linearised to* A, written $L \sqsubseteq_{\mathcal{R}} A$, if $A \subseteq \mathcal{H}^{\mathsf{seq}}_{\Theta,\Theta'}$ and for any $h \in [\![L]\!]$ there exists $h' \in A$ such that $h \sqsubseteq_{\mathcal{R}} h'$. We write $L \sqsubseteq_{\mathcal{R}} L'$ if $L \sqsubseteq_{\mathcal{R}} [\![L']\!] \cap \mathcal{H}^{\mathsf{seq}}_{\Theta,\Theta'}$.

▶ **Example 13.** We consider a higher-order variant of an example from [3] that motivates relational linearisability. Flat combining [11] is a synchronisation paradigm that advocates

the use of a single thread holding a global lock to process requests of all other threads. To facilitate this, threads share an array to which they write the details of their requests and wait either until they acquire a lock or their request has been processed by another thread. Once a thread acquires a lock, it executes all requests stored in the array and the outcomes are written to the array for access by the requesting threads.

Let $\Theta' = \{run \in \mathsf{Meths}_{(\theta\to\theta')\times\theta,\theta'}\}$. The library $L_{\mathsf{fc}} : \varnothing \to \Theta'$ (Figure 3, right) is built following the flat combining approach and, on acquisition of the global lock, the winning thread acts as a combiner of all registered requests. Note that the requests will be attended to one after another (thus guaranteeing mutual exclusion) and only one lock acquisition will suffice to process one array of requests. Using our framework, one can show that L_{fc} can be $\mathcal{R}$-linearised to the specification given by the library L_{spec} defined by

$$run = \lambda\,(f,x).\ (lock.\,acquire\,();\ \ \mathsf{let}\ \ result\ = f(x)\ \ \mathsf{in}\ \ lock.\,release\,();\ \ result\,)$$

where each function call in L_{spec} is protected by a lock. Observe that we cannot hope for $L_{\mathsf{fc}} \sqsubseteq L_{\mathsf{spec}}$, because clients may call library methods with functional arguments that recognise thread identity. Consequently, we can relate the two libraries only if context behaviour is guaranteed to be independent of thread identifiers. This can be expressed through $\sqsubseteq_{\mathcal{R}}$, where $\mathcal{R} \subseteq \mathcal{H}_{\varnothing,\Theta'} \times \mathcal{H}_{\varnothing,\Theta'}$ is a relation capturing thread-blind client behaviour.

3 Library syntax

We now look at the concrete syntax of libraries and clients. Libraries comprise collections of typed methods whose argument and result types adhere to the grammar: $\theta ::= \mathsf{unit} \mid \mathsf{int} \mid \theta \to \theta \mid \theta \times \theta$.

We shall use three disjoint enumerable sets of names, referred to as Vars, Meths and Refs, to name respectively variables, methods and references. x, f (and their decorated variants) will be used to range over Vars; m will range over Meths; and r over Refs. Methods and references are implicitly typed, i.e. $\mathsf{Meths} = \biguplus_{\theta,\theta'} \mathsf{Meths}_{\theta,\theta'}$ and $\mathsf{Refs} = \mathsf{Refs}_{\mathsf{int}} \uplus \biguplus_{\theta,\theta'} \mathsf{Refs}_{\theta,\theta'}$, where $\mathsf{Meths}_{\theta,\theta'}$ contains names for methods of type $\theta \to \theta'$, $\mathsf{Refs}_{\mathsf{int}}$ contains names of integer references and $\mathsf{Refs}_{\theta,\theta'}$ contains names for references to methods of type $\theta \to \theta'$. We write $\uplus$ for disjoint set union.

The syntax for libraries and clients is given in Figure 4. Each library L begins with a series of method declarations (public or abstract) followed by a block B containing method implementations ($m = \lambda x.M$) and reference initialisations ($r := i$ or $r := \lambda x.M$). The typing rules ensure that each public method is implemented within the block, in contrast to abstract methods. Clients are parallel compositions of closed terms.

Terms M specify the shape of allowable method bodies. $()$ is the skip command, i ranges over integers, t_{id} is the current thread identifier and $\oplus$ represents standard arithmetic operations. Thanks to higher-order references, we can simulate divergence by $(!r)()$, where $r \in \mathsf{Refs}_{\mathsf{unit},\mathsf{unit}}$ is initialised with $\lambda x^{\mathsf{unit}}.(!r)()$. Similarly, $\mathsf{while}\ M\ N$ can be simulated by $(!r)()$ after $r := \lambda x^{\mathsf{unit}}.\mathsf{let}\ y = M\ \mathsf{in}\ (\mathsf{if}\ y\ \mathsf{then}\ (N; (!r)())\ \mathsf{else}\ ())$. We also use the standard derived syntax for sequential composition, i.e. $M; N$ stands for $\mathsf{let}\ x = M\ \mathsf{in}\ N$, where x does not occur in N. For each term M, we write $\mathsf{Meths}(M)$ for the set of method names occurring in M. We use the same notation for method names in blocks and libraries.

▶ Remark 14. In Section 2 we used lock-related operations in our example libraries (*acquire*, *tryacquire*, *release*), on the understanding that they can be coded using shared memory. Similarly, the array of Example 13 in the sequel can be constructed using references.

$$\begin{array}{llll} \textit{Libraries} & L ::= B \mid \mathsf{abstract}\ m;\ L \mid \mathsf{public}\ m;\ L & \textit{Clients} & K ::= M \,\|\cdots\|\, M \\ \textit{Blocks} & B ::= \epsilon \mid m = \lambda x.M;\ B \mid r := \lambda x.M;\ B \mid r := i;\ B & \textit{Values} & v ::= () \mid i \mid m \mid \langle v, v\rangle \\ \textit{Terms} & M ::= () \mid i \mid t_{\mathsf{id}} \mid x \mid m \mid M \oplus M \mid \langle M, M\rangle \mid \pi_1\, M \mid \pi_2\, M \mid \mathsf{if}\ M\ \mathsf{then}\ M\ \mathsf{else}\ M & & \\ & \quad \mid \lambda x^{\theta}.M \mid xM \mid mM \mid \mathsf{let}\ x = M\ \mathsf{in}\ M \mid r := M \mid\, !r & & \end{array}$$

$$\frac{}{\Gamma \vdash () : \mathsf{unit}} \quad \frac{}{\Gamma \vdash i : \mathsf{int}} \quad \frac{}{\Gamma \vdash t_{\mathsf{id}} : \mathsf{int}} \quad \frac{\Gamma(x) = \theta}{\Gamma \vdash x : \theta} \quad \frac{m \in \mathsf{Meths}_{\theta,\theta'}}{\Gamma \vdash m : \theta \to \theta'} \quad \frac{\Gamma \vdash M : \mathsf{int} \quad \Gamma \vdash M_0, M_1 : \theta}{\Gamma \vdash \mathsf{if}\ M\ \mathsf{then}\ M_1\ \mathsf{else}\ M_0 : \theta}$$

$$\frac{\Gamma \vdash M : \theta_1 \times \theta_2}{\Gamma \vdash \pi_i\, M : \theta_i \ \ (i = 1,2)} \quad \frac{\Gamma \vdash M_i : \theta_i \ \ (i = 1,2)}{\Gamma \vdash \langle M_1, M_2\rangle : \theta_1 \times \theta_2} \quad \frac{\Gamma \vdash M_1, M_2 : \mathsf{int}}{\Gamma \vdash M_1 \oplus M_2 : \mathsf{int}} \quad \frac{\Gamma, x : \theta \vdash M : \theta'}{\Gamma \vdash \lambda x^{\theta}.M : \theta \to \theta'}$$

$$\frac{\Gamma(x) = \theta \to \theta' \quad \Gamma \vdash M : \theta}{\Gamma \vdash xM : \theta'} \quad \frac{m \in \mathsf{Meths}_{\theta,\theta'} \quad \Gamma \vdash M : \theta}{\Gamma \vdash mM : \theta'} \quad \frac{\Gamma \vdash M : \theta \quad \Gamma, x : \theta \vdash N : \theta'}{\Gamma \vdash \mathsf{let}\ x = M\ \mathsf{in}\ N : \theta'}$$

$$\frac{r \in \mathsf{Refs}_{\mathsf{int}} \quad \Gamma \vdash M : \mathsf{int}}{\Gamma \vdash r := M : \mathsf{unit}} \quad \frac{r \in \mathsf{Refs}_{\theta,\theta'} \quad \Gamma \vdash M : \theta \to \theta'}{\Gamma \vdash r := M : \mathsf{unit}} \quad \frac{r \in \mathsf{Refs}_{\mathsf{int}}}{\Gamma \vdash\, !r : \mathsf{int}} \quad \frac{r \in \mathsf{Refs}_{\theta,\theta'}}{\Gamma \vdash\, !r : \theta \to \theta'}$$

$$\frac{}{\vdash_{\mathsf{B}} \epsilon : \varnothing} \quad \frac{m \in \mathsf{Meths}_{\theta,\theta'} \quad x : \theta \vdash M : \theta' \quad \vdash_{\mathsf{B}} B : \Theta}{\vdash_{\mathsf{B}} m = \lambda x.M;\ B : \Theta \uplus \{m\}} \quad \frac{r \in \mathsf{Refs}_{\theta,\theta'} \quad x : \theta \vdash M : \theta' \quad \vdash_{\mathsf{B}} B : \Theta}{\vdash_{\mathsf{B}} r := \lambda x.M;\ B : \Theta}$$

$$\frac{r \in \mathsf{Refs}_{\mathsf{int}} \quad \vdash_{\mathsf{B}} B : \Theta}{\vdash_{\mathsf{B}} r := i;\ B : \Theta} \quad \frac{\vdash_{\mathsf{B}} B : \Theta}{\mathsf{Meths}(B) \vdash_{\mathsf{L}} B : \varnothing \to \Theta} \quad \frac{\Theta \uplus \{m\} \vdash_{\mathsf{L}} L : \Theta' \to \Theta'' \quad m \in \Theta''}{\Theta \vdash_{\mathsf{L}} \mathsf{public}\ m; L : \Theta' \to \Theta''}$$

$$\frac{\Theta \uplus \{m\} \vdash_{\mathsf{L}} L : \Theta' \to \Theta'' \quad m \notin \Theta''}{\Theta \vdash_{\mathsf{L}} \mathsf{abstract}\ m; L : \Theta' \uplus \{m\} \to \Theta''} \quad \frac{\vdash M_j : \mathsf{unit} \ \ (j = 1, \cdots, N)}{\Theta \vdash_{\mathsf{K}} M_1 \,\|\cdots\|\, M_N : \mathsf{unit}} \ \forall j.\, \mathsf{Meths}(M_j) \subseteq \Theta$$

Figure 4 Library syntax, and typing rules for terms ($\vdash$), blocks ($\vdash_{\mathsf{B}}$), libraries ($\vdash_{\mathsf{L}}$), clients ($\vdash_{\mathsf{K}}$).

For simplicity, we do not include private methods, yet the same effect could be achieved by storing them in higher-order references. As we explain in the next section, references present in library definitions are de facto private to the library. Note also that, according to our definition, sets of abstract and public methods are disjoint. However, given $m, m' \in \mathsf{Refs}_{\theta,\theta'}$, one can define a "public abstract" method with: $\mathsf{public}\ m;\ \mathsf{abstract}\ m';\ m = \lambda x^{\theta}.m'x$.

Terms are typed in environments $\Gamma = \{x_1 : \theta_1, \cdots, x_n : \theta_n\}$. Method blocks are typed through judgements $\vdash_{\mathsf{B}} B : \Theta$, where $\Theta \subseteq \mathsf{Meths}$. The judgments collect the names of methods defined in a block as well as making sure that the definitions respect types and are not duplicated. Also, the initialisation statements must comply with types.

Finally, we type libraries using statements of the form $\Theta \vdash_{\mathsf{L}} L : \Theta' \to \Theta''$, where $\Theta, \Theta', \Theta'' \subseteq \mathsf{Meths}$ and $\Theta' \cap \Theta'' = \varnothing$. The judgment $\varnothing \vdash_{\mathsf{L}} L : \Theta' \to \Theta''$ guarantees that any method occurring in L is present either in Θ' or Θ'', that all methods in Θ' are declared as abstract and unimplemented, while all methods in Θ'' are declared as public and defined. Thus, $\varnothing \vdash_{\mathsf{L}} L : \Theta \to \Theta'$ is a library in which Θ, Θ' are the abstract and public methods respectively. In this case, we also write $L : \Theta \to \Theta'$.

4 Semantics and soundness

The semantics of our system is given in several stages. First, we define an operational semantics for sequential and concurrent terms that may draw methods from a repository. We then adapt it to capture interactions of concurrent clients with closed libraries (no abstract methods). This notion is then used to define contextual approximation for arbitrary libraries. Finally, we introduce a trace semantics of arbitrary libraries, which generates the histories on which our notions of linearisability are based.

$$(L) \longrightarrow_{\text{lib}} (L, \varnothing, S_{\text{init}}) \qquad (r := i; B, \mathcal{R}, S) \longrightarrow_{\text{lib}} (B, \mathcal{R}, S[r \mapsto i])$$
$$(\mathsf{abstract}\ m; L, \mathcal{R}, S) \longrightarrow_{\text{lib}} (L, \mathcal{R}, S) \qquad (m = \lambda x.M; B\,, \mathcal{R}, S) \longrightarrow_{\text{lib}} (B, \mathcal{R}_{**}, S)$$
$$(\mathsf{public}\ m; L, \mathcal{R}, S) \longrightarrow_{\text{lib}} (L, \mathcal{R}, S) \qquad (r := \lambda x.M; B, \mathcal{R}, S) \longrightarrow_{\text{lib}} (B, \mathcal{R}_{**}, S[r \mapsto m])$$

$$(E[t_{\mathsf{id}}], \mathcal{R}, S) \to_t (E[t], \mathcal{R}, S) \qquad (E[\mathsf{if}\ i_*\ \mathsf{then}\ M_1\ \mathsf{else}\ M_0], \mathcal{R}, S) \to_t (E[M_{j_*}], \mathcal{R}, S)$$
$$(E[i_1 \oplus i_2], \mathcal{R}, S) \to_t (E[i_{**}], \mathcal{R}, S) \qquad (E[\pi_j\langle v_1, v_2\rangle], \mathcal{R}, S) \to_t (E[v_j], \mathcal{R}, S)$$
$$(E[!r], \mathcal{R}, S) \to_t (E[S(r)], \mathcal{R}, S) \qquad (E[\mathsf{let}\ x = v\ \mathsf{in}\ M], \mathcal{R}, S) \to_t (E[M\{v/x\}], \mathcal{R}, S)$$
$$(E[\lambda x.M], \mathcal{R}, S) \to_t (E[m], \mathcal{R}_{**}, S) \qquad (E[mv], \mathcal{R}_*, S) \to_t (E[M\{v/x\}], \mathcal{R}_*, S)$$

$$E ::= \bullet \mid E \oplus M \mid i \oplus E \mid \mathsf{if}\ E\ \mathsf{then}\ M\ \mathsf{else}\ M \mid \pi_j\, E \mid \langle E, M\rangle \mid \langle v, E\rangle \mid mE \mid \mathsf{let}\ x = E\ \mathsf{in}\ M \mid r := E$$

$$\frac{(M, \mathcal{R}, S) \to_t (M', \mathcal{R}', S')}{(M_1 \| \cdots \| M_{t-1} \| M \| M_{t+1} \| \cdots \| M_N, \mathcal{R}, S) \Longrightarrow (M_1 \| \cdots \| M_{t-1} \| M' \| M_{t+1} \| \cdots \| M_N, \mathcal{R}', S')}\ (K_N)$$

Figure 5 Evaluation rules for libraries ($\longrightarrow_{\text{lib}}$), terms ($\to_t$) and clients ($\Longrightarrow$). In the rules above we use the conditions/notation: $\mathcal{R}_{**} = \mathcal{R} \uplus (m \mapsto \lambda x.M)$, $i_{**} = i_1 \oplus i_2$, $\mathcal{R}_*(m) = \lambda x.M$, and $j_* = 0$ iff $i_* = 0$.

4.1 Library-client evaluation

Libraries, terms and clients are evaluated in environments comprising:

- A method environment $\mathcal{R}$, called *own-method repository*, which is a finite partial map on Meths assigning to each m in its domain, with $m \in \mathsf{Meths}_{\theta,\theta'}$, a term of the form $\lambda y.M$ (we omit type-superscripts from bound variables for economy).
- A finite partial map $S : \mathsf{Refs} \rightharpoonup (\mathbb{Z} \cup \mathsf{Meths})$, called *store*, which assigns to each r in its domain an integer (if $r \in \mathsf{Refs}_{\mathsf{int}}$) or name from $\mathsf{Meths}_{\theta,\theta'}$ (if $r \in \mathsf{Refs}_{\theta,\theta'}$).

The evaluation rules are presented in Figure 5, where we also define *evaluation contexts* E.

▶ Remark 15. We shall assume that reference names used in libraries are library-private, i.e. sets of reference names used in different libraries are assumed to be disjoint. Similarly, when libraries are being used by client code, this is done on the understanding that the references available to that code do not overlap with those used by libraries. Still, for simplicity, we shall rely on a single set Refs of references in our operational rules.

First we evaluate the library to create an initial repository and store. This is achieved by the first set of rules in Figure 5, where we assume that S_{init} is empty. Thus, library evaluation produces a tuple $(\epsilon, \mathcal{R}_0, S_0)$ including a method repository and a store, which can be used as the initial repository and store for evaluating $M_1 \| \cdots \| M_N$ using the (K_N) rule. We shall call the latter evaluation semantics for clients (denoted by $\Longrightarrow$) the *multi-threaded operational semantics*. The latter relies on closed-term reduction ($\to_t$), whose rules are given in the middle group, where t is the current thread index. Note that the rules for $E[\lambda x.M]$ in the middle group, along with those for $m = \lambda x.M$ and $r := \lambda x.M$ in the first group, involve the creation of a fresh method name m, which is used to put the function in the repository $\mathcal{R}$. Name creation is non-deterministic: any fresh m of the appropriate type can be chosen.

We define termination for clients linked with libraries that have no abstract methods. Recall our convention (Remark 15) that L and $M_1, \cdots, M_N$ must access disjoint parts of the store. Terms $M_1, \cdots, M_N$ can share reference names, though.

▶ **Definition 16.** Let $L : \varnothing \to \Theta'$ and $\Theta' \vdash_{\mathsf{K}} M_1 \| \cdots \| M_N : \mathsf{unit}$. We say that $M_1 \| \cdots \| M_N$ *terminates with linked library* L if $(M_1 \| \cdots \| M_N, \mathcal{R}_0, S_0) \Longrightarrow^* ((\) \| \cdots \| (\), \mathcal{R}, S)$, for some $\mathcal{R}, S$, where $(L) \longrightarrow^*_{\mathrm{lib}} (\epsilon, \mathcal{R}_0, S_0)$. We then write $\mathsf{link}\ L\ \mathsf{in}\ (M_1 \| \cdots \| M_N) \Downarrow$.

We shall build a notion of contextual approximation of libraries on top of termination: one library will be said to approximate another if, whenever the former terminates when composed with any parameter library and client, so does the latter.

We will be considering the following notions for composing libraries. Let us denote a library L as $L = D; B$, where D contains all the (public/abstract) method declarations of L, and B is its method block. We write $\mathsf{Refs}(L)$ for the set of references in L. Let $L_1 : \Theta_1 \to \Theta_2$ be of the form $D_1; B_1$. Given $L_2 : \Theta_1' \to \Theta_2'$ $(= D_2; B_2)$ such that $\Theta_2 \cap \Theta_2' = \mathsf{Refs}(L_1) \cap \mathsf{Refs}(L_2) = \varnothing$, $\Theta = \{m_1, \cdots, m_n\} \subseteq \Theta_2$ and $L' : \varnothing \to \Theta_1, \Theta'$, we define the *union* of L_1 and L_2, the Θ-*hiding* of L_1, and the *sequencing* of L' with L_1 respectively as:

$$\begin{array}{ll} L_1 \cup L_2 : (\Theta_1 \cup \Theta_1') \setminus (\Theta_2 \cup \Theta_2') \to \Theta_2 \cup \Theta_2' & = (D_1; B_1) \cup (D_2; B_2) = D_1'; D_2'; B_1; B_2 \\ L_1 \setminus \Theta : \Theta_1 \to (\Theta_2 \setminus \Theta) & = (D_1; B_1) \setminus \Theta = D_1''; B_1'\{!r_1/m_1\} \cdots \{!r_n/m_n\} \\ L'; L_1 : \varnothing \to \Theta_2, \Theta' & = (L' \cup L_1) \setminus \Theta_1 \end{array}$$

where D_1' is D_1 with any $\mathsf{abstract}\, m$ declaration removed for $m \in \Theta_2'$, dually for D_2'; and where D_1'' is D_1 without $\mathsf{public}\ m$ declarations for $m \in \Theta$ and each r_i is a fresh reference matching the type of m_i, and B_1' is obtained from B_1 by replacing each $m_i = \lambda x.M$ by $r_i := \lambda x.M$. Thus, the union of L_1 and L_2 corresponds to merging their code and removing any $\mathsf{abstract}$ declarations for methods that become defined. The hiding of a public method simply renders it private via the use of references.

▶ **Definition 17.** Given $L_1, L_2 : \Theta \to \Theta'$, we say that L_1 ***contextually approximates*** L_2, written $L_1 \sqsubseteq L_2$, if for all $L' : \varnothing \to \Theta, \Theta''$ and $\Theta', \Theta'' \vdash_{\mathsf{K}} M_1 \| \cdots \| M_N : \mathsf{unit}$, if $\mathsf{link}\ L'; L_1\ \mathsf{in}\ (M_1 \| \cdots \| M_N) \Downarrow$ then $\mathsf{link}\ L'; L_2\ \mathsf{in}\ (M_1 \| \cdots \| M_N) \Downarrow$. In this case, we also say that L_2 ***contextually refines*** L_1.

Note that, according to this definition, the parameter library L' may communicate directly with the client terms through a common interface Θ''. We shall refer to this case as the *general* case. Later on, we shall also consider more restrictive testing scenarios in which this possibility of explicit communication is removed. Moreover, from the disjointness conditions in the definitions of sequencing and linking we have that L_i, L' and $M_1 \| \cdots \| M_N$ access pairwise disjoint parts of the store.

4.2 Trace semantics

Building on the earlier semantics, we next introduce a trace semantics of libraries in the spirit of game semantics [2]. As mentioned in Section 2, the behaviour of a library will be represented as an exchange of moves between two players called P and O, representing the library and its corresponding context respectively. The context consists of the client of the library as well as the parameter library, with an index on each move ($\mathcal{K}/\mathcal{L}$) specifying which of them is involved in the move.

In contrast to the previous section, we handle scenarios in which called methods need not be present in the repository $\mathcal{R}$. Calls to such undefined methods are represented by labelled transitions – calls to the context made on behalf of the library (P). The calls can later be responded to with labelled transitions corresponding to returns, made by the context (O). On the other hand, O is able to invoke methods in $\mathcal{R}$, which will also be represented through suitable labels. Because we work in a higher-order setting, calls and returns made by both players may involve methods as arguments or results. Such methods also become available

(Int) $(\mathcal{E}, M, \mathcal{R}, \mathcal{P}, \mathcal{A}, S) \rightarrow_t (\mathcal{E}, M', \mathcal{R}', \mathcal{P}, \mathcal{A}, S')$, given that $(M, \mathcal{R}, S) \rightarrow_t (M', \mathcal{R}', S')$ and $\mathsf{dom}(\mathcal{R}' \setminus \mathcal{R})$ consists of names that do not occur in $\mathcal{E}, \mathcal{A}$.

(PQy) $(\mathcal{E}, E[mv], \mathcal{R}, \mathcal{P}, \mathcal{A}, S) \xrightarrow{\mathsf{call}\ m(v')_{PY}}_t (m :: E :: \mathcal{E}, -, \mathcal{R}', \mathcal{P}', \mathcal{A}, S)$, given $m \in \mathcal{A}_Y$ and (**PC**).

(OQy) $(\mathcal{E}, -, \mathcal{R}, \mathcal{P}, \mathcal{A}, S) \xrightarrow{\mathsf{call}\ m(v)_{OY}}_t (m :: \mathcal{E}, M\{v/x\}, \mathcal{R}, \mathcal{P}, \mathcal{A}', S)$, given $m \in \mathcal{P}_Y$, $\mathcal{R}(m) = \lambda x.M$ and (**OC**).

(PAy) $(m :: \mathcal{E}, v, \mathcal{R}, \mathcal{P}, \mathcal{A}, S) \xrightarrow{\mathsf{ret}\ m(v')_{PY}}_t (\mathcal{E}, -, \mathcal{R}', \mathcal{P}', \mathcal{A}, S)$, given $m \in \mathcal{P}_Y$ and (**PC**).

(OAy) $(m :: E :: \mathcal{E}, -, \mathcal{R}, \mathcal{P}, \mathcal{A}, S) \xrightarrow{\mathsf{ret}\ m(v)_{OY}}_t (\mathcal{E}, E[v], \mathcal{R}, \mathcal{P}, \mathcal{A}', S)$, given $m \in \mathcal{A}_Y$ and (**OC**).

(PC) If v contains the names $m_1, \cdots, m_k$ then $v' = v\{m_i'/m_i \mid 1 \leq i \leq k\}$ with each m_i' being a fresh name. Moreover, $\mathcal{R}' = \mathcal{R} \uplus \{m_i' \mapsto \lambda x.m_i x \mid 1 \leq i \leq k\}$ and $\mathcal{P}' = \mathcal{P} \cup_Y \{m_1', \cdots, m_k'\}$.

(OC) If v contains names $m_1, \cdots, m_k$ then $m_i \in \phi(\mathcal{P}, \mathcal{A})$, for each i, and $\mathcal{A}' = \mathcal{A} \cup_Y \{m_1, \cdots, m_k\}$.

Figure 6 Trace semantics rules. The rule (Int) is for embedding internal rules. In the rule (PQy), the library (P) calls one of its abstract methods (either the original ones or those acquired via interaction), while in (PAy) it returns from such a call. The rules (OQy) and (OAy) are dual and represent actions of the context. In all of the rules, whenever we write $m(v)$ or $m(v')$, we assume that the type of v matches the argument type of m.

for future calls: function arguments/results supplied by P are added to the repository and can later be invoked by O, while function arguments/results provided by O can be queried in the same way as abstract methods.

The trace semantics utilises configurations that carry more components than the previous semantics. We define two kinds of configurations:

$$\textit{O-configurations}\ (\mathcal{E}, -, \mathcal{R}, \mathcal{P}, \mathcal{A}, S) \quad \text{and} \quad \textit{P-configurations}\ (\mathcal{E}, M, \mathcal{R}, \mathcal{P}, \mathcal{A}, S)$$

where the component $\mathcal{E}$ is an *evaluation stack*, that is, a stack of the form $[X_1, X_2, \cdots, X_n]$ with each X_i being either an evaluation context or a method name. On the other hand, $\mathcal{P} = (\mathcal{P}_\mathcal{L}, \mathcal{P}_\mathcal{K})$ with $\mathcal{P}_\mathcal{L}, \mathcal{P}_\mathcal{K} \subseteq \mathsf{dom}(\mathcal{R})$ being sets of *public* method names, and $\mathcal{A} = (\mathcal{A}_\mathcal{L}, \mathcal{A}_\mathcal{K})$ is a pair of sets of *abstract* method names. $\mathcal{P}$ will be used to record all the method names produced by P and passed to O: those passed to $O\mathcal{K}$ are stored in $\mathcal{P}_\mathcal{K}$, while those leaked to $O\mathcal{L}$ are kept in $\mathcal{P}_\mathcal{L}$. Inside $\mathcal{A}$, the story is the opposite one: $\mathcal{A}_\mathcal{K}$ ($\mathcal{A}_\mathcal{L}$) stores the method names produced by $O\mathcal{K}$ (resp. $O\mathcal{L}$) and passed to P. Consequently, the sets of names stored in $\mathcal{P}_\mathcal{L}, \mathcal{P}_k, \mathcal{A}_\mathcal{L}, \mathcal{A}_k$ will always be disjoint.

Given a pair $\mathcal{P}$ as above and a set $Z \subseteq \mathsf{Meths}$, we write $\mathcal{P} \cup_\mathcal{K} Z$ for the pair $(\mathcal{P}_\mathcal{L}, \mathcal{P}_\mathcal{K} \cup Z)$. We define $\cup_\mathcal{L}$ in a similar manner, and extend it to pairs $\mathcal{A}$ as well. Moreover, given $\mathcal{P}$ and $\mathcal{A}$, we let $\phi(\mathcal{P}, \mathcal{A})$ be the set of *fresh* method names for $\mathcal{P}, \mathcal{A}$: $\phi(\mathcal{P}, \mathcal{A}) = \mathsf{Meths} \setminus (\mathcal{P}_\mathcal{L} \cup \mathcal{P}_\mathcal{K} \cup \mathcal{A}_\mathcal{L} \cup \mathcal{A}_\mathcal{K})$.

We give the rules generating the trace semantics in Figure 6. Note that the rules are parameterised by: P/O and Y, which together determine the polarity of the next move; Q/A, which stands for the move being a call (*Question*) or a return (*Answer*) respectively. The rules depict the intuition presented above. When in an O-configuration, the context may issue a call to a public method $m \in \mathcal{P}_Y$ and pass control to the library (rule (OQy)). Note that, when this occurs, the name m is added to the evaluation stack $\mathcal{E}$ and a P-configuration is obtained. From there on, the library will compute internally using rule (Int), until: it either needs to evaluate an abstract method (i.e. some $m' \in \mathcal{A}_Y$), and hence issues a call via rule (PQy); or it completes its computation and returns the call (rule (PAy)). Calls to abstract methods, on the other hand, are met either by further calls to public methods (via (OQy)), or by returns (via (OAy)).

Finally, we extend the trace semantics to a concurrent setting where a fixed number of N-many threads run in parallel. Each thread has separate evaluation stack and term

components, which we write as $\mathcal{C} = (\mathcal{E}, X)$ (where X is a term or "$-$"). Thus, a configuration now is of the following form:

$$N\text{-}\textit{configuration}\ \ (\mathcal{C}_1 \| \cdots \| \mathcal{C}_N, \mathcal{R}, \mathcal{P}, \mathcal{A}, S)$$

where, for each i, $\mathcal{C}_i = (\mathcal{E}_i, X_i)$ and $(\mathcal{E}_i, X_i, \mathcal{R}, \mathcal{P}, \mathcal{A}, S)$ is a sequential configuration. We shall abuse notation a little and write $(\mathcal{C}_i, \mathcal{R}, \mathcal{P}, \mathcal{A}, S)$ for $(\mathcal{E}_i, X_i, \mathcal{R}, \mathcal{P}, \mathcal{A}, S)$. Also, below we write $\vec{\mathcal{C}}$ for $\mathcal{C}_1 \| \cdots \| \mathcal{C}_N$ and $\vec{\mathcal{C}}[i \mapsto \mathcal{C}'] = \mathcal{C}_1 \| \cdots \| \mathcal{C}_{i-1} \| \mathcal{C}' \| \mathcal{C}_{i+1} \| \cdots \| \mathcal{C}_N$ and, for economy, we use $\mathcal{RPAS}$ to range over tuples $(\mathcal{R}, \mathcal{P}, \mathcal{A}, S)$. The concurrent traces are produced by the following two rules

$$\frac{(\mathcal{C}_i, \mathcal{RPAS}) \to_i (\mathcal{C}', \mathcal{RPAS}')}{(\vec{\mathcal{C}}, \mathcal{RPAS}) \Longrightarrow (\vec{\mathcal{C}}[i \mapsto \mathcal{C}'], \mathcal{RPAS}')}\ (\textsc{PInt}) \qquad \frac{(\mathcal{C}_i, \mathcal{RPAS}) \xrightarrow{x_{XY}}_i (\mathcal{C}', \mathcal{RPAS}')}{(\vec{\mathcal{C}}, \mathcal{RPAS}) \xRightarrow{(i,x)_{XY}} (\vec{\mathcal{C}}[i \mapsto \mathcal{C}'], \mathcal{RPAS}')}\ (\textsc{PExt})$$

with the proviso that the names freshly produced internally in (PInt) are fresh for the whole of $\vec{\mathcal{C}}$.

We can now define the trace semantics of a library L. We call a configuration component $\mathcal{C}_i$ ***final*** if it is in one of the following forms, for O- and P-configurations respectively: $\mathcal{C}_i = ([\,], -)$ or $\mathcal{C}_i = ([\,], ())$. We call $(\vec{\mathcal{C}}, \mathcal{R}, \mathcal{P}, \mathcal{A}, S)$ final just if $\vec{\mathcal{C}} = \mathcal{C}_1 \| \cdots \| \mathcal{C}_N$ and each $\mathcal{C}_i$ is final.

▶ **Definition 18.** For each $L : \Theta \to \Theta'$, we define the N-trace semantics of L to be:

$$[\![L]\!]_N = \{\, s \mid (\vec{\mathcal{C}}_0, \mathcal{R}_0, (\varnothing, \Theta'), (\Theta, \varnothing), S_0) \xRightarrow{s}{}^{*} \rho \ \wedge\ \rho \text{ final} \,\}$$

where $\vec{\mathcal{C}}_0 = ([\,], -) \| \cdots \| ([\,], -)$ and $(L) \longrightarrow^{*}_{\text{lib}} (\epsilon, \mathcal{R}_0, S_0)$. We may write $[\![L]\!]_N$ simply as $[\![L]\!]$.

We are now able to revisit the linearisability claims anticipated in Examples 1, 11 and 13.

▶ **Lemma 19** (Linearisability examples)**.**

1. $L_{\mathsf{mset}} \sqsubseteq A_{\mathsf{mset}}$,
2. $L_{\mathsf{mset2}} \sqsubseteq_{\text{enc}} A_{\mathsf{mset2}}$,
3. $L_{\mathsf{fc}} \sqsubseteq_{\mathcal{R}} L_{\mathsf{spec}}$.

We conclude the presentation of the trace semantics by providing a semantics for library contexts. Recall that in our setting (Figure 1) a library $L : \Theta \to \Theta'$ is deployed in a context consisting of a parameter library $L' : \varnothing \to \Theta, \Theta''$ and a concurrent composition of client threads $\Theta', \Theta'' \vdash M_i : \mathsf{unit}$ $(i = 1, \cdots, N)$. We shall write $\mathsf{link}\ L'\mathsf{;}-\ \mathsf{in}\ (M_1 \| \cdots \| M_N)$, or simply C, to refer to such contexts.

▶ **Definition 20.** Let $\Theta', \Theta'' \vdash_{\mathsf{K}} M_1 \| \cdots \| M_N : \mathsf{unit}$ and $L' : \varnothing \to \Theta, \Theta''$. We define:

$$[\![\mathsf{link}\ L'\mathsf{;}-\ \mathsf{in}\ (M_1 \| \cdots \| M_N)]\!] = \{\, s \mid (\vec{\mathcal{C}}_0, \mathcal{R}_0, (\Theta, \varnothing), (\varnothing, \Theta'), S_0) \xRightarrow{s}{}^{*} \rho \wedge \rho \text{ final} \,\}$$

where $(L') \longrightarrow^{*}_{\text{lib}} (\epsilon, \mathcal{R}_0, S_0)$ and $\vec{\mathcal{C}}_0 = ([\,], M_1) \| \cdots \| ([\,], M_N)$.

▶ **Lemma 21.** *For any $L : \Theta \to \Theta'$, $L' : \varnothing \to \Theta, \Theta''$ and $\Theta', \Theta'' \vdash_{\mathsf{K}} M_1 \| \cdots \| M_N : \mathsf{unit}$ we have $[\![L]\!]_N \subseteq \mathcal{H}_{\Theta,\Theta'}$ and $[\![\mathsf{link}\ L'\mathsf{;}-\ \mathsf{in}\ (M_1 \| \cdots \| M_N)]\!] \subseteq \mathcal{H}^{co}_{\Theta,\Theta'}$.*

4.3 Soundness

To conclude, we clarify in what sense all the notions of linearisability are sound. Recall the general notion of contextual approximation (refinement) from Definition 17. In the encapsulated case libraries are being tested by clients that do not communicate with the parameter library explicitly. The corresponding definition of contextual approximation is defined below.

▶ **Definition 22 (Encapsulated $\sqsubseteq$).** Given libraries $L_1, L_2 : \Theta \to \Theta'$, we write $L_1 \sqsubseteq_{\text{enc}} L_2$ when, for all $L' : \varnothing \to \Theta$ and $\Theta' \vdash_{\mathsf{K}} M_1 \| \cdots \| M_N : \mathsf{unit}$, if $\mathsf{link}\ L'\,;L_1\ \mathsf{in}\ (M_1\|\cdots\|M_N)\Downarrow$ then $\mathsf{link}\ L'\,;L_2\ \mathsf{in}\ (M_1\|\cdots\|M_N)\Downarrow$.

For relational linearisability, we need yet another notion that will link $\mathcal{R}$ to contextual testing.

▶ **Definition 23.** Let $\mathcal{R} \subseteq \mathcal{H}_{\Theta,\Theta'} \times \mathcal{H}_{\Theta,\Theta'}$ be a set closed under permutation of names in $\mathsf{Meths} \setminus (\Theta \cup \Theta')$. We say that a context formed by L' and $M_1, \cdots, M_N$ is $\mathcal{R}$-closed if, for any $h \in [\![\mathsf{link}\ L'\,;-\ \mathsf{in}\ (M_1\|\cdots\|M_N)]\!]$, $\overline{h}\,\mathcal{R}\,\overline{h'}$ implies $h' \in [\![\mathsf{link}\ L'\,;-\ \mathsf{in}\ (M_1\|\cdots\|M_N)]\!]$. Given $L_1, L_2 : \Theta \to \Theta'$, we write $L_1 \sqsubseteq_{\mathcal{R}} L_2$ if, for all $\mathcal{R}$-closed contexts formed from $L', M_1, \cdots, M_N$, whenever $\mathsf{link}\ L'\,;L_1\ \mathsf{in}\ (M_1\|\cdots\|M_N)\Downarrow$ then we also have $\mathsf{link}\ L'\,;L_2\ \mathsf{in}\ (M_1\|\cdots\|M_N)\Downarrow$.

▶ **Theorem 24** (Correctness)**.**

1. *$L_1 \sqsubseteq L_2$ implies $L_1 \precsim L_2$.*
2. *$L_1 \sqsubseteq_{\text{enc}} L_2$ implies $L_1 \precsim_{enc} L_2$.*
3. *$L_1 \sqsubseteq_{\mathcal{R}} L_2$ implies $L_1 \precsim_{\mathcal{R}} L_2$.*

Finally, linearisability is compatible with library composition. $\sqsubseteq$ is closed under union with libraries that use disjoint stores, while $\sqsubseteq_{\text{enc}}$ is closed under a form of sequencing that respects encapsulations.

5 Related and future work

Linearisability has been consistently used as a correctness criterion for concurrent algorithms on a variety of data structures [18], and has inspired a variety of proof methods [5]. An explicit connection between linearisability and refinement was made in [6], where it was shown that, in base-type settings, linearisability and refinement coincide. Similar results have been proved in [4, 9, 17, 3]. Our contributions are notions of linearisability that serve as correctness criteria for libraries with methods of arbitrary order and have a similar relationship to refinement. The next natural target is to investigate proof methods for establishing linearisability of higher-order concurrent libraries. The examples proved herein are only an initial step in that direction.

At the conceptual level, [6] proposed that the verification goal behind linearisability is observational refinement. In this vein, [24] utilised logical relations as a direct method for proving refinement in a higher-order concurrent setting, while [23] introduced a program logic that builds on logical relations. On the other hand, proving conformance to a history specification has been addressed in [20] by supplying history-aware interpretations to off-the-shelf Hoare logics for concurrency. Other logic-based approaches for concurrent higher-order libraries, which do not use linearisability, include Higher-Order and Impredicative Concurrent Abstract Predicates [21, 22].

Acknowledgements. We thank the authors of [3] for bringing the higher-order linearisability problem to our attention, Radha Jagadeesan and Kasper Svendsen for constructive comments, and C. Tzevelekou for help with Figure 1.

References

1. http://c-cube.github.io/ocaml-containers/0.21/CCMultiSet.S.html.
2. S. Abramsky and G. McCusker. Game semantics. In H. Schwichtenberg and U. Berger, editors, *Logic and Computation*. Springer-Verlag, 1998. Proceedings of the 1997 Marktoberdorf Summer School.
3. A. Cerone, A. Gotsman, and H. Yang. Parameterised linearisability. In *Proceedings of ICALP'14*, volume 8573 of *Lecture Notes in Computer Science*, pages 98–109. Springer, 2014.
4. J. Derrick, G. Schellhorn, and H. Wehrheim. Mechanically verified proof obligations for linearizability. *ACM Trans. Program. Lang. Syst.*, 33(1):4, 2011.
5. B. Dongol and J. Derrick. Verifying linearisability: A comparative survey. *ACM Comput. Surv.*, 48(2):19, 2015.
6. I. Filipovic, P. W. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52):4379–4398, 2010.
7. D. R. Ghica and A. S. Murawski. Angelic semantics of fine-grained concurrency. In *Proceedings of FOSSACS*, volume 2987 of *Lecture Notes in Computer Science*, pages 211–225. Springer-Verlag, 2004.
8. D. R. Ghica and N. Tzevelekos. A system-level game semantics. *Electr. Notes Theor. Comput. Sci.*, 286:191–211, 2012.
9. A. Gotsman and H. Yang. Liveness-preserving atomicity abstraction. In *Proceedings of ICALP*, volume 6756 of *Lecture Notes in Computer Science*, pages 453–465. Springer-Verlag, 2011.
10. S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer III, and N. Shavit. A lazy concurrent list-based set algorithm. In *Proceedings of OPODIS*, volume 3974 of *Lecture Notes in Computer Science*, pages 3–16, Springer-Verlag, 2006.
11. D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of SPAA*, pages 355–364, ACM, 2010.
12. M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
13. R. Jagadeesan, G. Petri, C. Pitcher, and J. Riely. Quarantining weakness - compositional reasoning under relaxed memory models (extended abstract). In *Proceedings of ESOP*, volume 7792 of *Lecture Notes in Computer Science*, pages 492–511, Springer-Verlag, 2013.
14. A. Jeffrey and J. Rathke. A fully abstract may testing semantics for concurrent objects. *Theor. Comput. Sci.*, 338(1-3):17–63, 2005.
15. J. Laird. A game semantics of Idealized CSP. In *Proceedings of MFPS'01*, pages 1–26. Elsevier, 2001. ENTCS, Vol. 45.
16. J. Laird. A fully abstract trace semantics for general references. In *Proceedings of ICALP*, volume 4596 of *Lecture Notes in Computer Science*, pages 667–679. Springer, 2007.
17. H. Liang and X. Feng. Modular verification of linearizability with non-fixed linearization points. In Proceedings of PLDI, pages 459–470, ACM, 2013.
18. M. Moir and N. Shavit. Concurrent data structures. In *Handbook of Data Structures and Applications*. Chapman and Hall/CRC, 2004.
19. P. W. O'Hearn, N. Rinetzky, M. T. Vechev, E. Yahav, and G. Yorsh. Verifying linearizability with hindsight. In *Proceedings of PODC*, pages 85–94, ACM, 2010.
20. I. Sergey, A. Nanevski, and A. Banerjee. Specifying and verifying concurrent algorithms with histories and subjectivity. In *Proceedings of ESOP*, volume 9032 of *Lecture Notes in Computer Science*, pages 333–358, Springer, 2015.
21. K. Svendsen and L. Birkedal. Impredicative concurrent abstract predicates. In *Proceedings of ESOP*, volume 8410 of *Lecture Notes in Computer Science*, pages 149–168, Springer, 2014.

22 K. Svendsen, L. Birkedal, and M. J. Parkinson. Joins: A case study in modular specification of a concurrent reentrant higher-order library. In *Proceedings of ECOOP*, volume 7920 of *Lecture Notes in Computer Science*, pages 327–351, Springer, 2013.

23 A. Turon, D. Dreyer, and L. Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *Proceedings of ICFP*, pages 377–390, ACM, 2013.

24 A. J. Turon, J. Thamsborg, A. Ahmed, L. Birkedal, and D. Dreyer. Logical relations for fine-grained concurrency. In *Proceedings of POPL*, pages 343–356, ACM, 2013.